

Timing Studies for Efficient Data Transfer Between the NOvA Data Concentrator Modules and the NOvA Trigger Processors

Stephen Foulkes
Fermi National Accelerator Laboratory

Abstract

This document describes performance testing and results for the NOvA data acquisition development effort. The tests attempt to benchmark the performance of the Linux TCP/IP and SCTP/IP stack in an effort to optimize the software and hardware for the NOvA trigger processors.

1 Test System Setup

Four nodes in the FCC 2 computer room were used for this test. Each was a dual AMD Athlon 1900+ with 1GB of ram. The tests made use of the 3com 3c905c 10/100 Ethernet cards that were integrated with the motherboard. Each card had an MTU of 1500 bytes and was plugged into the same 10/100Mb switch.

All systems were identical, running Fermi Linux 4.2 with a 2.6.9 SMP kernel. The test software was compiled with the GNU C++ compiler, version 3.4.4. The only modification made to the test nodes was to open a port in the firewall so that test nodes could receive data.

2 Test System Software

Three different versions of the test system software have been written. The first uses TCP exclusively, and does not reassemble transmissions if they are broken up during transport. Its main purpose is to measure CPU usage as a function of the number of open connections. The second version also uses TCP, but reassembles transmissions if they are broken up during transport. The third version uses SCTP, which guarantees that message boundaries are kept intact.

The times() function was used to determine the run time of the program as well as the amount of processor time used. This is the same method that the “time” utility uses.

2.1 *Data Concentrator Module*

The software that simulated the Data Concentrator Module would open up a series of connections to a trigger processor node and then randomly select one of the open connections and send a stream of data. The number of connections, total number of transmissions, amount of data sent per transmission and the rate of transmissions are specified on the command line.

The second and third versions of the DCM software also include a 4 byte header that indicates the length of the message being sent.

2.2 *Trigger Processor Software*

The Trigger Processor software listens for and then accepts connections from the Data Concentrator Module. As data is sent, it is copied into a buffer and then passed off to the event building software. The Trigger Processor software is instrumented to keep track of

the total run time, the total CPU time used, the number of iterations in the main loop, as well as the number of connections that are handled in each iteration of the main loop. The total number of transmissions, total number of connections and the size of each transmission are passed to the Trigger Processor software from the command line.

The Trigger Processor software is a single threaded process, and uses the epoll interface available in the Linux 2.6 kernel to manage its open connections. Epoll is very similar to select and poll, in which it monitors a number of connections and then determines which connections are able to be read from or written to. What makes epoll better than select() or poll() is that the mechanism for passing down the list of connections to monitor is separate from the mechanism used to get the state of each connection. With epoll the connection list does not have to be passed to the kernel on every iteration of the main loop.

2.3 Python Scripts

Two python scripts were written to control the Data Concentrator Module and the Trigger Processor. A range of options would be specified in each script, and the script would take care of iterating through each option and logging the results.

Another python script was written to generate the test set for the second and third versions of the test software. A test set would consist of a list of message sizes that the DCM would send to the Trigger Processor.

3 Tests Run

Two instances of the first test were run: one with ~1900 transmissions per second and one with ~3800 transmissions per second. Each set of parameters was repeated four times during each instance. The number of open connections varied from 250 to 1000 in increments of 50. Six transmission sizes consisting of 10 bytes, 100 bytes, 250 bytes, 500 bytes, 1000 bytes and 2000 bytes were run for each set of open connections. A transmission size of 4000 bytes was also run for the test with ~1900 transmissions per second. This could not be run for the other test due to the fact that only 100Mb equipment was used.

For the second test, one million message sizes were generated by the testgen.py script. The average message size was about 9000 bytes, with a standard deviation of 1000 bytes. The script did not generate any message sizes that were less than 8000 bytes. The distribution can be seen in figure 1. A test was run with the amount of connections open varying from 250 to 500 in increments of 50. During these tests, the software did not do anything to limit the data rate as in previous tests.

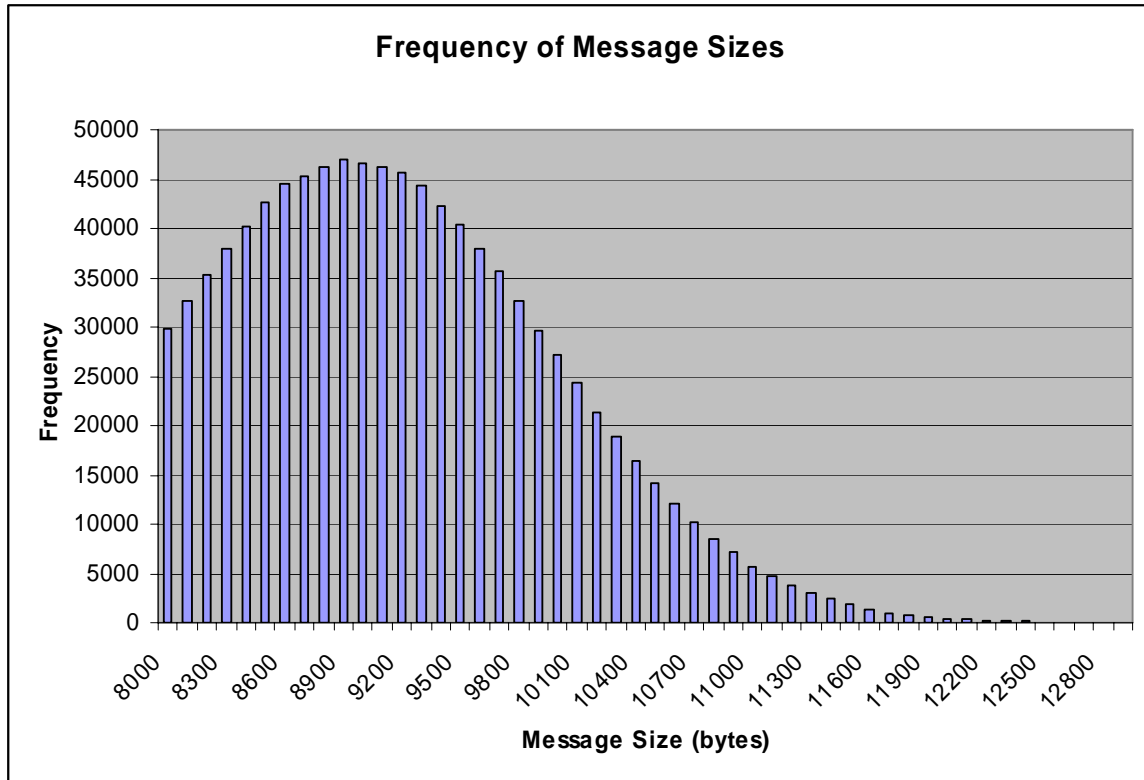


Figure 1

4 Test Results

4.1 *The number of open connections has little impact on CPU usage*

Figure 2 plots the number of open connections vs. CPU usage for the test run with ~1900 transmissions per second and figure 3 plots the same for the test run with ~3800 transmissions per second.

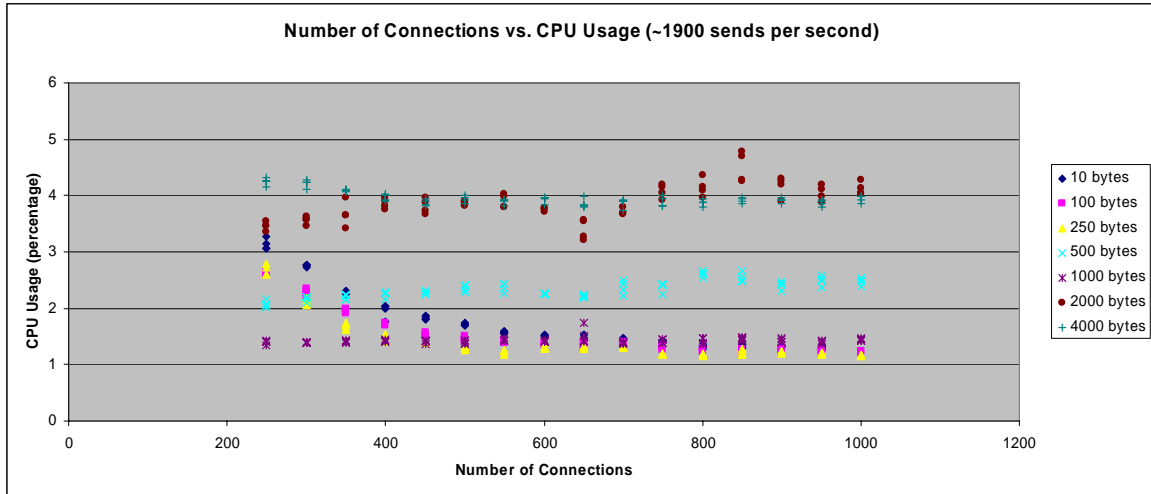


Figure 2

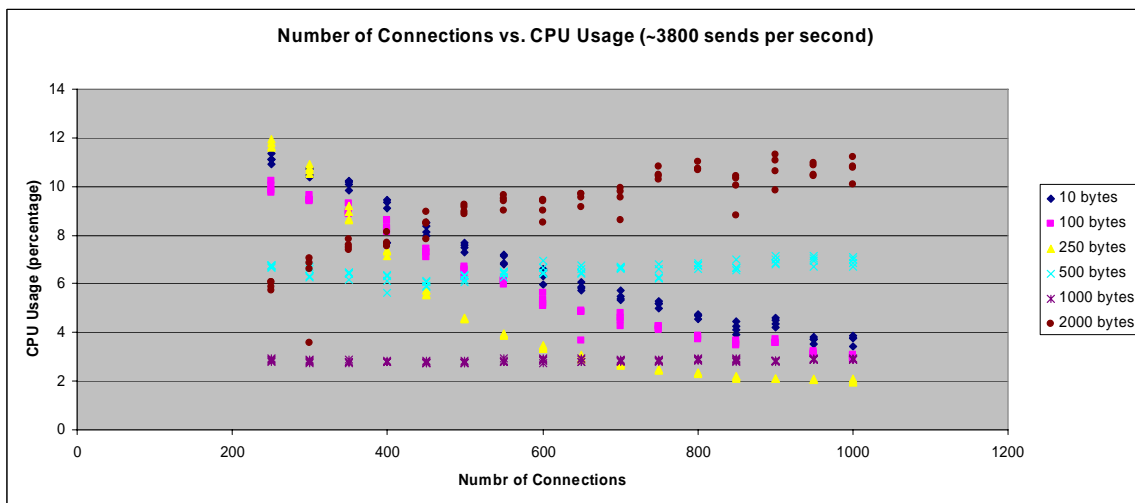


Figure 3

All tests run with transmission sizes of 500 and 1000 bytes are completely flat across the range of open connections. Tests run with smaller transmissions actually perform better with a larger number of connections.

The 10, 100 and 250 bytes tests experienced a factor of 3 less CPU usage with 1000 open connections then with 250 open connections. Figures 4 and 5 plot the CPU usage vs. the number of connections handled per loop iteration. These show that CPU usage increases as the number of connections handed per loop iteration decreases. This makes sense, as less loop iterations means less system calls and jumps between user and kernel space.

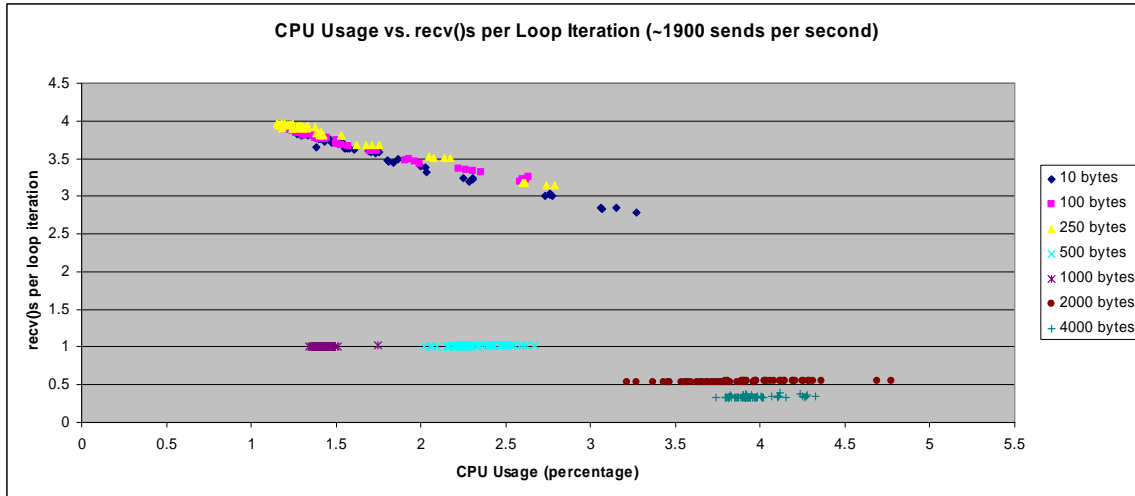


Figure 4

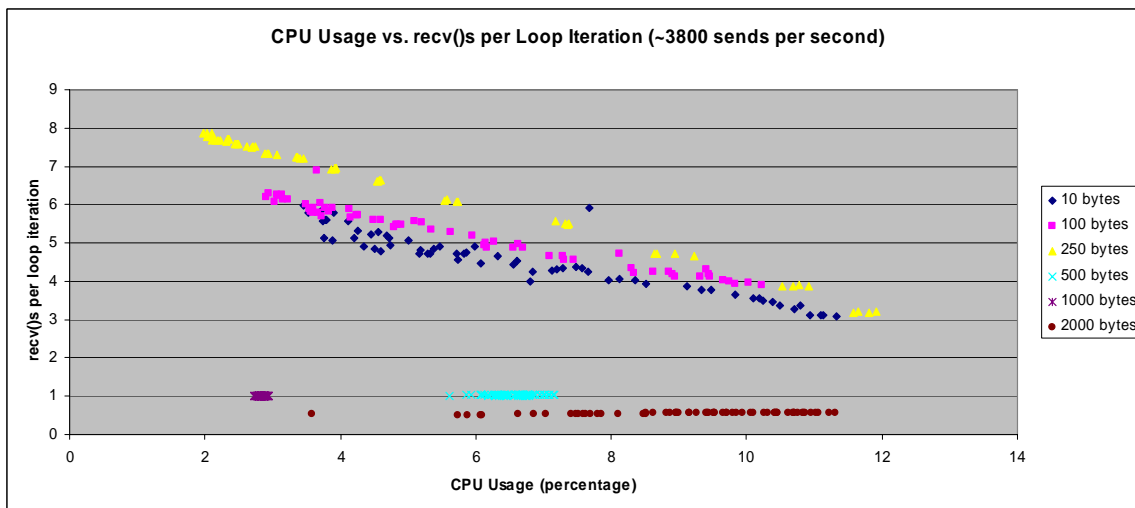


Figure 5

4.2 SCTP has more CPU overhead, as well as transfer overhead than TCP

Figure 6 plots the number of open connections vs. CPU usage for the tests that compared TCP to SCTP. On average, the SCTP test program required 30-40 percent more CPU than the TCP program to perform the same task.

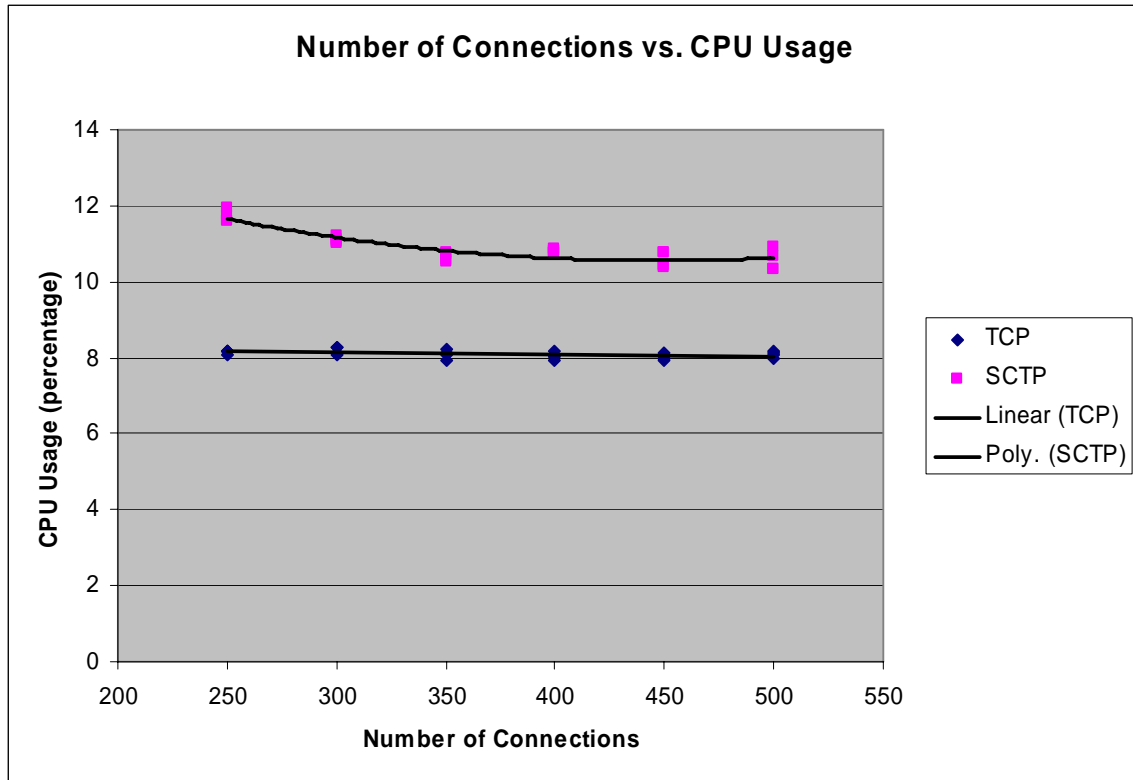


Figure 6

Figure 7 plots the number of open connections vs. the transfer rate for each test. The SCTP tests ran about 30% slower than the TCP tests due to increased overhead, and degraded with the number of open connections.

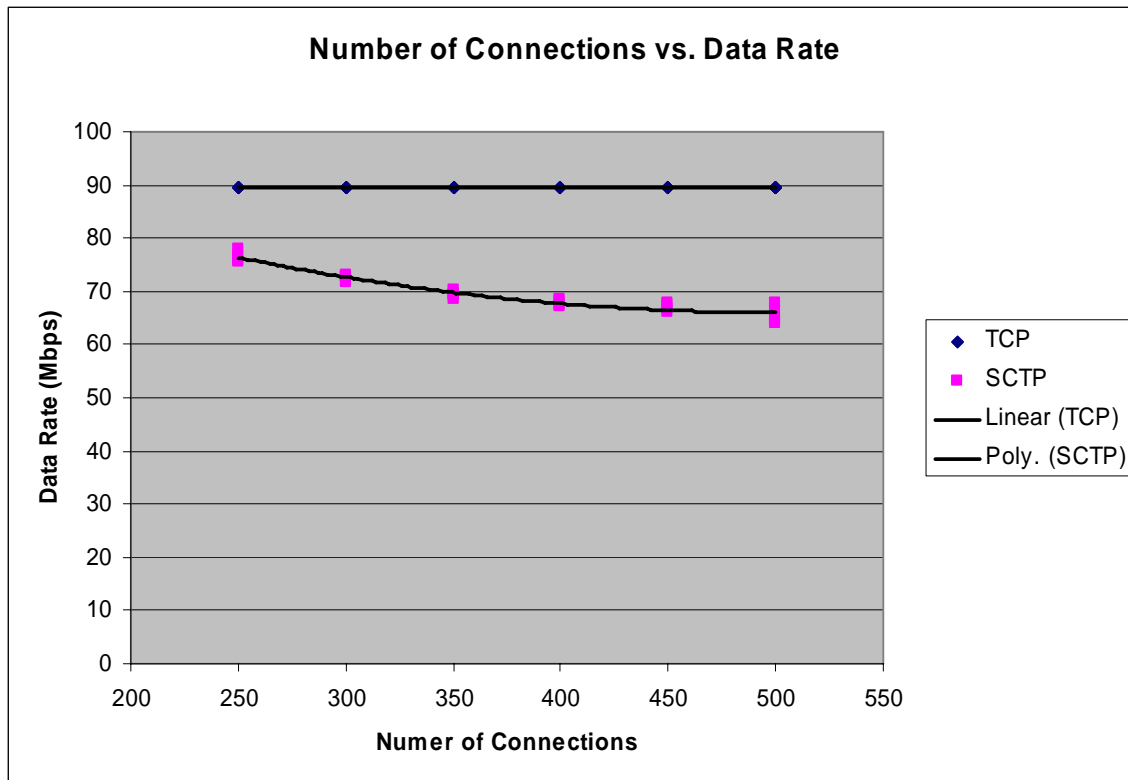


Figure 7

5 Apparent problems with SCTP

Initially, I had problems with the SCTP test programs stalling out in the middle of the tests. A quick search of the internet turned up several similar reports, and no apparent solutions. The source of my problems weren't with SCTP, but with the way I was polling the open sockets.

There are two ways to poll sockets with epoll: edge triggered and level triggered. In edge triggered mode, sockets are only returned from the `epoll_wait()` system call when their status changes. If a socket has 2kb of data waiting, and I only read 1kb, a subsequent call to `epoll_wait()` will not return this socket again. In level triggered mode, all sockets will be returned that have data waiting.

I initially had epoll set in edge triggered mode, and this was causing the hang-ups that I was seeing. I don't know why I didn't see this problem with the TCP test programs, as they were setup the same way. Changing epoll to run in level triggered mode solved the problem.